# Characterizing process change using log file data

*Mark Guzdial*
GVU Center, College of Computing, Georgia Institute of Technology
Atlanta, GA 30332-0280, (404) 853-9387, email: guzdial@cc.gatech.edu

*Chris Walton*
Advanced Technology Group, Professional Education Division, Arther Andersen & Co.
1405 North Fifth Avenue, St. Charles, IL 60174

*Michael Konemann*
Computer Science Department, Carroll College
Waukesha, WI 53186

*Elliot Soloway*
HiCE Research Group, Artificial Intelligence Laboratory, The University of Michigan
1101 Beal Avenue, Ann Arbor, MI 48109-2110, email: soloway@eecs.umich.edu

**ABSTRACT**
Students learning a new task with an unfamiliar interface must learn the task, the interface, and a task-to-device mapping which enables them to develop an efficient process for achieving goals with that software. To characterize student process and learning, we have used two methods which rely on log file data rather than the more typical interview-based data. The first method creates a graphic snapshot of process, and the second method creates a transition diagram of process. Both techniques are presented with examples of their use.

**KEYWORDS:** Evaluation methodology, educational software, learning, usability evaluation, log file data, event trace records, automated analyses

**INTRODUCTION**
The goal of many of the software development projects in the Highly Interactive Computing Environments (HiCE) Research Group at the University of Michigan is to teach students a process which can lead to successful performance of an unfamiliar, software-based task (e.g., multimedia composition, programming). We take the user's *process* with software to be the actions taken by students in the software interface and the order in which the actions are taken. Successful software leads students from their initial, naive process to a more efficient process. Evaluating the success of our software requires us to characterize user process and to measure change in student process.

We view the task of evaluating how students learn to use software to be an instance of the general problem of evaluating software usability. We can characterize the student's problem in terms of a general user's development of process. A user's process with software (i.e., the actions taken and the order in which they are taken) is dependent on three factors (based on Kieras and Polson [11]):

- The user's task representation,
- The user's device (interface) representation, and
- The correspondence between the user's goals and the methods available in the software, called the *task-to-device* mapping.

When users begin to use software with an unfamiliar interface, they must construct a device representation and a task-to-device mapping. If the users are also novice to the task (e.g., students when they first begin to program), they must also learn a task representation. As users develop this knowledge, we expect their process to change. For example, users will most probably reorder their task goals and subgoals over time to correspond to a more efficient use of the available methods in the software. The evidence for the reordered task goals will be different orderings of user actions. If the observed user's process does not change (e.g., become more efficient), we have reason to call into question the usability of the software (e.g., the interface may be too complex for the user to construct either the device representation or the task-to-device mapping) or the user's understanding of the task (e.g., perhaps additional support for understanding the domain is needed.) Thus, characterizing process and process change can provide valuable clues to the user's developing understanding.

Typically, an evaluation of user process would involve think-aloud protocols or interviews – some method

which involves querying the user to determine task understanding, interface understanding, and the task-to-device mapping [3]. A comparison over time should show a change in the user's process in terms of mapping and order of actions. However, protocols are labor-intensive (and thus difficult to do for a whole classroom of students) and intrusive (e.g., classroom structure does not easily permit a student thinking aloud while working). A method of characterizing process that would be more amenable to the realities of educational software would be to log user actions in the interface and analyze these data, which we call *log file data* (sometimes also called *event trace records* [2]). Log file data do not provide insight into task and interface representations, but they can provide valuable clues in terms of the observable performance. Specifically, we can address two kinds of questions about individual users and about classes of users:

1. Is a user developing a more efficient process over time during use of the software?
2. Are some users learning an efficient process while others are not?

In other words, log file analysis methods can alert the developer to problems: If a user or group of users do not seem to be improving in use of the software. The methods described here do not identify what the problem is. The usability of the software can be questioned if a problem exists, since it may be that the designer did not create methods which could be easily understood and mapped to task goals and subgoals. However, the problem might also be due to the user's lack of understanding of the task, as is often the case with students learning the task with the software.

This paper presents two methods for characterizing process and process change through analysis of log file data in three sections. The next section discusses some of the log file analysis methods described in the literature and presents our methods. The following two sections present use of these methods in two pieces of educational software. We use the first software environment, GPCeditor, to explore a graphical method which we use for comparing process for a single student over time. We use the second environment, Emile, to explore a method useful for comparing between students. The final section of the paper summarizes and highlights some of the open questions of log file analysis methods.

## METHODS OF LOG FILE ANALYSES

Collecting log file data to address usability concerns is a data collection method that is growing in popularity (e.g., as noted in the INTERCHI'93 panel on the subject [14]). There are two main approaches to automated analyses of log file data: counting of key variables and characterization of process.

A relatively simple strategy for analyzing log file data is to simply count key variables in the log data. Several of the speakers at the INTERCHI'93 panel on software tools for usability mentioned using spreadsheets for implementing this strategy [14]. Card, Moran, and Newell also use this strategy [3] to test their predictions of usability. A detailed example is provided in Shute and Glaser [12] where they gathered 30 variables on various uses of their educational simulation software (e.g., the number of experiments run, the number of notebook entries made) to measure the usefulness of their software. The counting method can provide useful information on a user's process, but does not provide a method of representing the overall process, the choice of actions over time.

A more complex strategy is to use the log file data to characterize user process. The key example is Hammer and Rouse's [9] use of Markov analysis (a common analysis technique found in most finite mathematics texts, e.g. [10]) to characterize keystroke data in use of an editor. They expected to find differences in process for different classes of users working on different tasks. Unfortunately, they instead found that the individual differences swamped all group differences. They suggest that their focus on low-level actions (e.g., typing one character, deleting one character) led to this problem, and that a focus at a higher-level, that is, a more task-specific level (e.g., correct spelling errors, adding new material), might have resulted in more significant differences.

Our approach is just this: To define high-level groups of actions and to use transitions between groups to characterize user process. We have an advantage that Hammer and Rouse did not have in their 1979 of text editors – the software that we are evaluating has a graphical user interface. Such an interface has an implied hierarchy of actions. For our applications, a menu operation is higher-level than clicking on an object to select it, and selecting an object is higher-level than dragging a window. While the interface hierarchy is not always consistent (for example, we sometimes classify a menu action as low-level), it provides a starting place for classifying the task-specificity of an interface action.

In the following two sections, we provide examples of two methods for using high-level groups to characterize process in analysis of log file data.

- In the first method, a graphic process pattern is computed which depicts the transition between high-level and low-level states over time, which can be visually compared to another process. This method is particularly useful when a mixture of high and low-level states are desired in an analysis. Because the comparison is visual, this method is most appropriate for exploring change for a single user over time.
- In the second method, Markov analysis is used to create transition diagrams that describe the observed probability of one high-level state following another. This method can be used

when all of the actions of interest in the analysis can be classified into high-level states. These diagrams can be compared across students using various quantitative methods.

The example of the first method will be a positive example, describing a user whose process becomes more efficient. The example of the second method will contrast a student's process that has become efficient with another student whose process has not.

## GPCEDITOR AND PROCESS PATTERNS

The GPCeditor (GoalPlanCode editor) is a Pascal programming environment which has been used by high school students for several years at a local high school [6,13]. Students using the GPCeditor do not type code directly. Rather they *decompose* the programming problem by defining goals (statements of program purpose) and plans (program segments for achieving a goal) and by relating these in a hierarchy. As the program components are decomposed, they can be *composed* into a complete program, and tested (*run*). The GPCeditor automatically collects log file data as the user works through the program.

In our analysis of the GPCeditor log file data, we defined three high-level states and three low-level states. The high-level states are defined in our model of process for the GPCeditor (see [13] for more on this model), and the low-level states were seen as key stepping stones in this process.

- *Decomposition* is a high-level state corresponding to creation or deletion actions of either goals or plans. Decomposition actions are all menu-based actions.
- *Composition* is a high-level state corresponding to assembly actions (e.g., inserting or removing a plan into a linear order). Composition actions are menu-based actions.
- *Run* is a high-level state corresponding to testing actions (e.g., running the program, tracing the program). Run actions are menu-based actions.
- *New day* is a low-level state corresponding to starting the GPCeditor.
- *Data objects* is a low-level state corresponding to creation and specification of data objects. Data objects are manipulated in a modal window, so data object actions are button-based actions.
- *Undo* is a low-level state corresponding to correcting previous actions.

We used an analysis tool called the Event Recorder [4,5] to draw process patterns based on these six states. The Event Recorder was developed by Berger and his colleagues to study patterns of student-teacher interactions in the classroom and to study navigation behavior in a hypermedia database. To prepare the data for the Event Recorder, we wrote a small program which recoded the original GPCeditor log files in terms of a time stamped series of these six events. The Event Recorder graphed each event at discrete vertical positions with time on the horizontal axis (see Figure 1 for an example). In our experience, this method is most appropriate when analyzing a mixture of both high and low-level states because the low-level states serve as contrast in the graph. A graph with all high-level states is difficult to read and interpret. (Recall that Hammer and Rouse found that studying all low-level states resulted in too great of individual variance.)

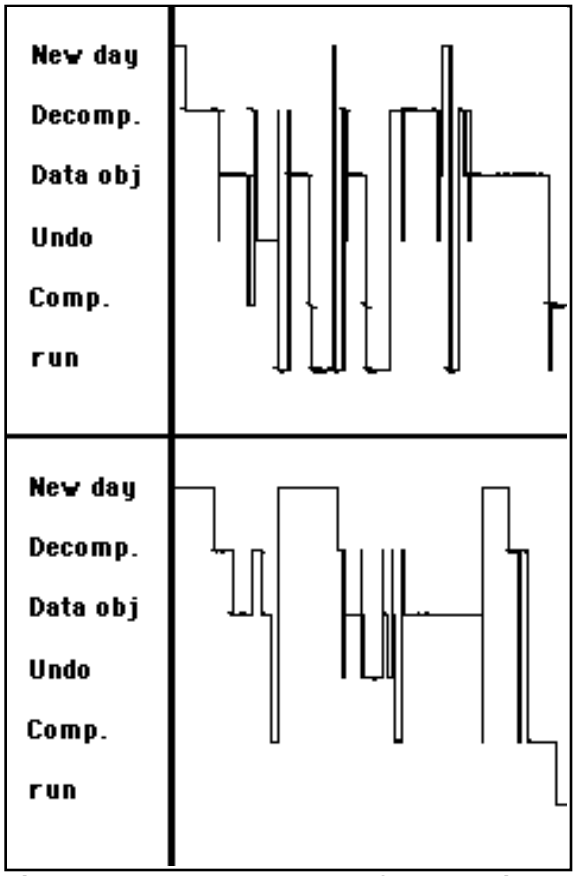


**Figure 1: Process patterns for a student using the GPCeditor**

Figure 1 shows the process patterns for a typical student during the first hour of work on two programs using the GPCeditor. The programs during this segment of the class ranged from 10 to 20 lines of code and usually took the students approximately two hours to complete.

- The first process pattern was taken from the fourth program undertaken by the student. This pattern shows frequent spikes down to the Run state. The student typically performs Decomposition actions, then Data object actions, then Composition actions, and finally Runs the program, before beginning again with Decomposition actions. This pattern is broken up with frequent use of Undo. This is a fairly inefficient process due to frequent tests of the program and frequent use of undo. (However, frequent tests of a program are good for a beginning student from a pedagogical point of view.)

- The second process pattern was taken from the ninth program undertaken by the student. This pattern features frequent repetitions of the Decomposition-Data-Composition cycle, but the student does not actually run the program until near the end of the hour (at which time the program did run correctly). Further, there is less use of undo. This is a process in which the student can create more complex programs in less time due to the efficient ordering and selection of software methods. (This pattern of postponed execution did become the prevalent pattern over time for students using the GPCeditor.)

Process patterns provide snapshot views of process which can be used to compare students over time. While this is not a quantitative method, it provides a level of insight into user actions and the ordering of these actions in a process which is not possible with simple counts of actions. As seen in Figure 1, it is possible to identify growing efficiency in user's process over time.

## EMILE AND PROCESS TRANSITION DIAGRAMS

Emile is a programming environment used by high school students to create simulations and demonstrations of physics concepts using graphical elements (e.g., buttons and text fields) as well as text program elements [8]. Like GPCeditor, Emile provides goals and program components which can be arranged in a hierarchy using *decomposition* actions. These components can be assembled into a complete program using *composition* actions, then tested using *debugging* actions. Unlike GPCeditor, Emile also provides prompts for student articulations, such as plans at the beginning of the day and a journal for summarizing the day's activities. Emile creates log files that record user actions.

Five high-level states were defined for analysis of Emile's log files, in order to use Markov analysis and avoid the Hammer and Rouse problem of low-level states causing high individual variance. All of the states were defined in terms of menu actions. In defining these high-level states, the low-level Undo action was not considered, and the New Day and Data Objects states were subsumed into high-level states.

- *Initial Review* is the default state at the beginning of a session and is also the state corresponding to initial review actions such as creating a plan or creating a project description.
- *Decomposition* is the state for actions that create goals or program components (e.g., a button, a field) and arrange them into hierarchies (e.g., collecting program components into groups, identifying a goal for a group).
- *Composition* is the state for actions that assemble the program into a whole (e.g., placing a button, referencing a field from a text program). In Emile, data object actions are considered to be part of the Composition state.
- *Debugging* is the state for actions that test the program.
- *Final Review* is the state for actions that naturally come at the end of a session or a project (e.g., creating a journal entry, storing components to a component library for later reuse).

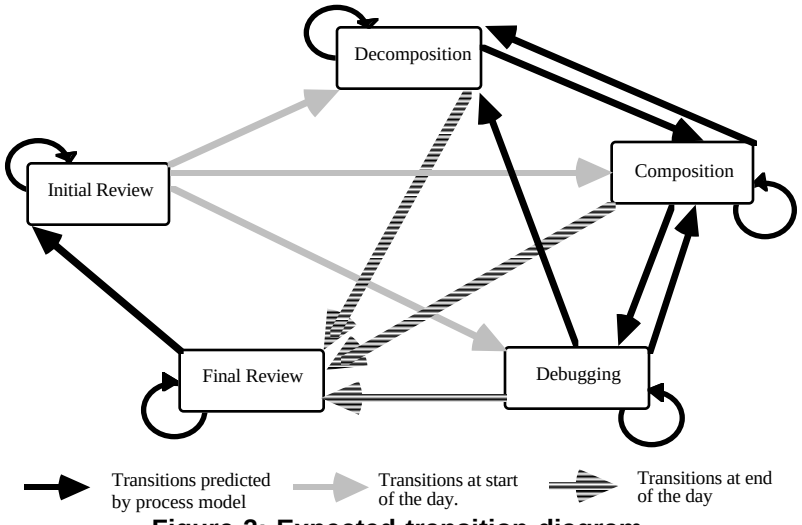


**Figure 2: Expected transition diagram**

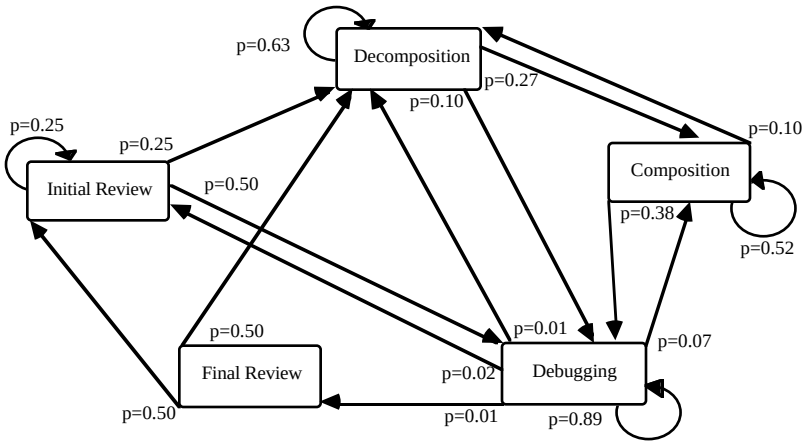


**Figure 3: Student M's transition diagram**

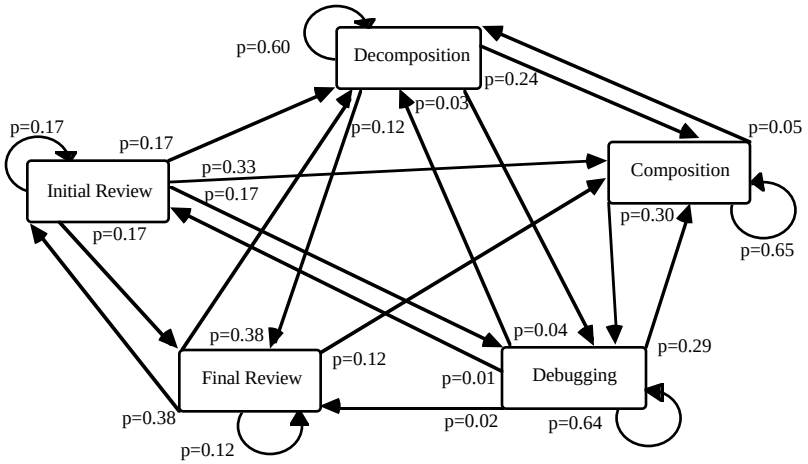


**Figure 4: Student C's transition diagram**

We analyzed Emile's log files using Hawk (Hypercard AWK), a text analysis tool based on an Awk-like programming language and implemented in a HyperCard environment (for more on Hawk, see [7]; for more on Awk, see [1]). We used Hawk to recode the Emile log files in terms of these five states, then to compute a transition diagram for each student's process on each program. While it should be possible to mix high-level and low-level states in a transition diagram, we avoided this to reduce individual variance.

Figure 2 describes the transition diagram that we considered efficient for use of Emile. There are 17 arcs in this diagram.

- The dark lines indicate transitions anticipated by the programming task. As seen in the GPCeditor process patterns, Decomposition actions are often followed by Composition actions which are often followed by Debugging actions (with some movement between these actions to correct bugs). After a Debugging action, the user might create a new component with a Decomposition action.
- The dotted lines indicate transitions expected after starting a new session. After starting Emile (perhaps creating a plan), students are expected to continue their programs where they left off.
- The dashed lines indicate transitions expected at the end of a session. Wherever students are in their programming task, students at the end of the day are asked to make a journal entry before ending (perhaps also saving components to their library).

Figure 3 and 4 are the transition diagrams for two students working on their fourth programs in Emile. Each arc is notated with the observed probability of that transition occurring. For example, the arc from Initial Review to Decomposition in Figure 3 annotated with "p=0.25" means that one out of every four Initial Review actions was followed by a Decomposition action.

- Figure 3 shows student M's transition diagram. This diagram is quite close to the expected diagram. There is an additional transition here from Final Review to Decomposition which is unusual: It means that he sometimes did a Final Review action (e.g., created a journal entry), then returned to a Decomposition action. Some arcs are missing, such as from Decomposition or Composition to Final Review. This only means that the student was

always Debugging before ending the session in Final Review. There are 16 arcs in this diagram.

- Figure 4 shows student C's transition diagram. Here, there are several more arcs than are expected. For example, the student has a transition from Initial Review to Final Review, suggesting that the student created a journal entry immediately after starting the program. There are also arcs from Final Review to both Decomposition and Composition which are unexpected. In total, there are 21 arcs in this diagram.

In general, we found that the number of transition diagram arcs was a reasonable indication of the fit between the student's transition diagram and the expected process. Clearly, counting arcs is simplistic and does not consider the observed probabilities at all, but the arc counts have highlighted important differences in groups of students. Hammer and Rouse used yet another method for quantifying transition diagram methods which involved computing a statistic indicating whether different chains of actions might indeed be from the same process [9]. We chose the count of transition diagram arcs as a simpler method that provided insight into our data.

Table 1 summarizes the number of transition diagram arcs for five students at the same level of experience (the fourth program in a study of Emile). Note that students B, M, and S are all at a relatively low number of transition diagrams arcs (15, 16, and 14, respectively), while students C and L are at a relatively high number of arcs (21 each), compared with the expected number of 17 transition diagram arcs. This is a case where the inefficient process used by students C and L may be due to a lack of task knowledge rather than a bad device design. In testing at the end of the study, students C and L also had the least understanding of programming and physics. Thus, their inability to create an efficient task-to-device mapping could be due to a lack of task knowledge rather than a design problem.

## SUMMARY

This paper has presented two methods for analyzing log file data in order to characterize process and process change, as opposed to use of interview-based methods. We see the exploration of user process and process change as providing critical clues about the user's learning of task, device (interface), and task-to-device mapping. Both methods are based on the definition of high-level states which allow for meaningful depictions of user process. The first method creates process patterns describing the transitions between both high-level and low-level states in a graphic, snapshot view. The second method uses Markov analysis to define transition diagrams which can be quantified and compared across students. Using automated and unobtrusive log file analyses provided us with two critical pieces of information:

- Evidence of student process learning and a characterization of how the process was changing.
- Evidence for differences in how different groups of students were learning software process.

Some of the many open questions about these methods and log file analysis methods in general include:

- What is a methodology for selecting states and actions of interest for log file analyses? We identified states based on our expected process models, and we favored actions corresponding to menu items and button presses before others. A more rigorous methodology is required to create general techniques.

- What is the difference between low-level and high-level states for log file analyses? Hammer and Rouse said that their states were too low-level, so we emphasized high-level states and found useful information. A clear definition of low-level and high-level states with an empirical exploration of where each is useful in log-file-based usability analysis is called for in order to explore this hierarchy of process states.
- What are the tradeoffs between log file analyses and interview-based analyses for characterizing process? We explored log file analyses out of necessity, because of the difficulties of using interviews and think-aloud protocols in a classroom. Our experience suggest that log file analysis is a more powerful technique for characterizing process than is currently reflected in the literature. A more careful analysis is needed to determine the strengths and weaknesses of log file analyses as compared to other analysis techniques for characterizing process.

This research has been supported in part by NSF Contract MDR-9010362.

| Student | B | C | L | M | S |
|---|---|---|---|---|---|
| Number of arcs | 15 | 21 | 21 | 16 | 14 |

**Table 1: Number of transition diagram arcs for each student on Project 4**

## REFERENCES


1. Aho, A.V., Brian W. Kernighan, and Peter J. Weinberger. The AWK Programming Language. Reading, MA: Addison-Wesley, 1988.

2. Badre, Albert N., Scott E. Hudson, and Paulo J. Santos. An environment to support user interface evaluation using synchronized video and event trace recording. Georgia Institute of Technology, GVU Center, 1993. Technical Report GIT-GVU-93-16.

3. Card, Stuart K., Thomas P. Moran, and Allen Newell. The Pyschology of Human-Computer Interaction. Hillsdale, NJ: Lawrence Erlbaum and Associates, 1983.

4. Dershimer, Charles and Carl Berger. "Characterizing student interactions with a hypermedia learning environment." Paper presented at the American Educational Research Association annual meeting, San Francisco, CA., 1992.

5. Dershimer, Charles, Carl Berger, and David Jackson. "Designing hyper-media for concept development: Formative evaluation through analysis of log files." Paper presented at the National Association for Research in Science Teaching annual meeting, Fontana, WI., 1991.

6. Guzdial, Mark, Elliot Soloway, Phyllis Blumenfeld, Luke Hohmann, Ken Ewing, Iris Tabak, Kathy Brade, and Yasmin Kafai. "The future of CAD: Technological support for kids building artifacts." In Learning to design, designing to learn: Using technology to transform the curriculum, eds. D. Balestri, S. Ehrmann, and D.L. Ferguson. Norwood, NJ: Ablex Publishing Company, 1992.

7. Guzdial, Mark J. "Deriving software usage patterns from log files." Submitted to Behavior Research Methods, Instruments, and Computers (1993)

8. Guzdial, Mark J. "Emile: Software-realized scaffolding for science learners programming in mixed media." Unpublished Ph.D. dissertation, University of Michigan, 1993.

9. Hammer, J.M. and W.B. Rouse. "Analysis and modeling of freeform text editing behavior." In Proceedings of the 1979 International Conference on Cybernetics and Society, 659-664. Denver: 1979.

10. Kemeny, J.G., J.L. Snell, and G. L. Thompson. Introduction to Finite Mathematics. Englewood Cliffs, NJ: Prentice-Hall, 1974.

11. Kieras, David and Peter G. Polson. "An approach to the formal analysis of user complexity." International Journal of Man-Machine Studies 22 (1985): 365-394.

12. Shute, Valerie J. and Robert Glaser. "A large-scale evaluation of an intelligent discovery world: Smithtown." Interactive Learning Environments 1 (1, 1990): 51-77.

13. Soloway, Eliot, Mark Guzdial, Kathy Brade, Luke Hohmann, Iris Tabak, Peri Weingrad, and Phyllis Blumenfeld. "Technological support for the learning and doing of design." In Foundations and frontiers of adaptive learning environments, eds. Marlene Jones and P.H. Winne. New York: Springer-Verlag, 1993.

14. Weiler, Paul, Richard Cordes, Monty Hammontree, Derek Hoiem, and Michael Thompson. "Software for the usability lab: A sampling of current tools (Panel session)." In INTERCHI'93 Conference Proceedings: Conference on Human Factors in Computing Systems, eds. Stacey Ashlund, Kevin Mullet, Austin Henderson, Erik Hollnagel, and Ted White. New York: ACM, 1993.